10.3 A "Manager" Class

In the previous section, we defined four different data classes—`Restaurant`, `Customer`, `Courier`, `Order`—to represent different entities in our food delivery system. We must now determine how to keep track of all of these entities, and how they can interact with each other. For example, as a user I would want to be able to look up a list of restaurants in my area to order food from. In code, how does a single `Customer` object "know" about all the different `Restaurants` in the system? Should each `Customer` have an attribute containing list of `Restaurants`?[1]

[1] The question of how objects "know" about other objects is similar to the notion of variable *scope*. A variable's scope determines where it can be accessed in a program; the scope of an object dictates the object's lifetime and who the object belongs to. But now consider our current problem domain, with the hundreds of restaurants and potential thousands of customers. What should the scope of all those objects be?

There are many ways to approach this problem. A common object-oriented design approach is to create a new manager class whose role is to keep track of all of the entities in the system and to mediate the interactions between them (like a customer placing a new order). This class is more complex than the others we saw in the last section, and so we will not use a data class, and instead use a general class with a custom initializer and keep most of the instance attributes private.

Here is the manager class we'll create for our food delivery system. The `FoodDeliverySystem` class will store (and have access to) every customer, courier, and restaurant represented in our system.

```python
class FoodDeliverySystem:
    """A system that maintains all entities (restaurants, customers,
         couriers, and orders).

    Public Attributes:
        - name: the name of this food delivery system

    Representation Invariants:
        - self.name != ''
        - all(r == self._restaurants[r].name for r in self._restaurants)
        - all(c == self._customers[c].name for c in self._customers)
        - all(c == self._couriers[c].name for c in self._couriers)
    """
    name: str

    # Private Instance Attributes:
    #   - _restaurants: a mapping from restaurant name to Restaurant object.
    #       This represents all the restaurants in the system.
    #   - _customers: a mapping from customer name to Customer object.
    #       This represents all the customers in the system.
    #   - _couriers: a mapping from courier name to Courier object.
    #       This represents all the couriers in the system.
    #   - _orders: a list of all orders (both open and completed orders).

    _restaurants: dict[str, Restaurant]
    _customers: dict[str, Customer]
    _couriers: dict[str, Courier]
    _orders: list[Order]

    def __init__(self, name: str) -> None:
        """Initialize a new food delivery system with the given company name.

        The system starts with no entities.
        """
        self.name = name

        self._restaurants = {}
        self._customers = {}
        self._couriers = {}
        self._orders = []
```

Changing state

What we have done so far is model the *static* properties of our food delivery system, that is, the attributes that are necessary to capture a particular snapshot of the state of the system at a specific moment in time. Next, we're going to look at how to model the *dynamic* properties of the system: how the entities interact with each other and cause the system state to change over time.

Adding entities

Though a `FoodDeliverySystem` instance starts off empty, we can define simple methods to add entities to the system.[2] By making our collection attributes private and requiring client code call these methods, we can check for uniqueness of these entity names as well.

[2] You can picture this happening when a new restaurant/customer/courier signs up for our app.

```
class FoodDeliverySystem:
    ...

    def add_restaurant(self, restaurant: Restaurant) -> bool:
        """Add the given restaurant to this system.

        Do NOT add the restaurant if one with the same name already exists.

        Return whether the restaurant was successfully added to this system.
        """
        if restaurant.name in self._restaurants:
            return False
        else:
            self._restaurants[restaurant.name] = restaurant
            return True

    def add_customer(self, customer: Customer) -> bool:
        """Add the given customer to this system.

        Do NOT add the customer if one with the same name already exists.

        Return whether the customer was successfully added to this system.
        """
        # Similar implementation to add_restaurant

    def add_courier(self, courier: Courier) -> bool:
        """Add the given courier to this system.

        Do NOT add the courier if one with the same name already exists.

        Return whether the courier was successfully added to this system.
        """
        # Similar implementation to add_restaurant
```

Placing orders

The main driving force in our simulation is customer orders. When a customer places an order, a chain of events is triggered:

1. The order is sent to the restaurant and to the assigned courier.
2. The courier travels to the restaurant and picks up the food, and then brings it to the customer.
3. Once the courier has reached their destination, they indicate that the delivery has been made.

To represent these events in our program, we need to create functions that mutate the state of the system. Where should we create these functions? We could write them as top-level functions, or as methods of one of our existing entity classes (turning that class from a data class into a general class). We have previously said that one of the roles of the `FoodDeliverySystem` is to mediate interactions between the various entities in the system, and so this makes it a natural class to add these mutating methods.

```
class FoodDeliverySystem:
    ...

    def place_order(self, order: Order) -> None:
        """Record the new given order.
        Assign a courier to this new order (if a courier is available).

        Preconditions:
            - order not in self.orders
        """

    def complete_order(self, order: Order) -> None:
        """Mark the given order as complete.

        Make the courier who was assigned this order available to take a new
         order.

        Preconditions:
            - order in self.orders
        """
```

We could then place an order from a customer using `FoodDeliverySystem.place_order`, which would be responsible for both recording the order and assigning a courier to that order. `FoodDeliverySystem.complete_order` does the opposite, marking the order as complete and un-assigning the courier so that they are free to take a new order. With both `FoodDeliverySystem.place_order` and `FoodDeliverySystem.complete_order`, we can begin to see how a simulation might take place where many customers are placing orders to different restaurants that are being delivered by available couriers.

Note that this discussion should make sense even though we haven't implemented either of these methods. Questions like "How do we choose which courier to assign to a new order?" and "How do we mark an order as complete?" are about *implementation* rather than the public interface of these methods. We'll discuss one potential implementation of these methods in lecture, but we welcome you to attempt your own implementations as an exercise.